NPSCS-92-008

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Iterative Software Testing

Timothy Shimeall
Stephen Shimeall

June 1992

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared for the Naval Weapons Center and funded by the Naval Postgraduate School.

Timothy J. Shimeall
Assistant Professor
of Computer Science

Reviewed by:

Valdis Berzins
Associate Chairman for Research
Department of Computer Science

Released by:

PAUL J. MARTO
Dean of Research

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPSCS-92-008	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Postgraduate School; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: OM&N Direct Funding
8c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
ITERATIVE SOFTWARE TESTING(U)

12. PERSONAL AUTHOR(S)
Shimeall, Timothy J., Shimeall, Stephen C.l

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Progress	FROM 10/91 TO 6/92	1992, June 4	24

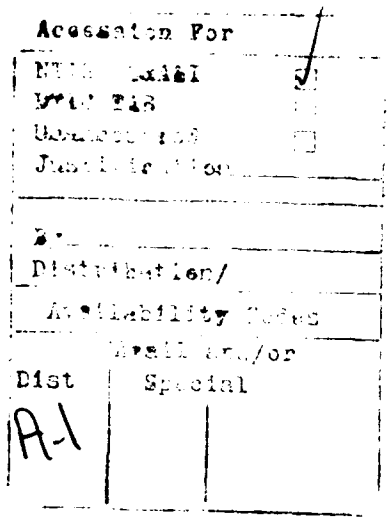
16. SUPPLEMENTARY NOTATION
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Iteration, Spiral Model, Audit, Evaluation, Test Planning,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software is often developed via a series of modifications and releases. This sort of development implies that testing often takes place in an increasingly familiar structure, rather than a unique or unfamiliar structure. This paper discusses how to exploit this type of development in test planning and execution, by presenting a motivational example, a formal model of test iteration and a discussion of tools to enhance the efficiency of testing when an iterative approach is used.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Timothy J. Shimeall	22b. TELEPHONE (Include Area Code): (408) 646-2509; 22c. OFFICE SYMBOL: CS/Sm



Iterative Software Testing

Timothy J. Shimeall[1]
Computer Science Department
Naval Postgraduate School
Monterey CA

Stephen C. Shimeall
Boeing Defense & Space Group
Boeing Aerospace
Seattle WA

Abstract

Software is often developed via a series of modifications and releases. This sort of development implies that testing often takes place in an increasingly familiar structure, rather than a unique or unfamiliar structure. This paper discusses how to exploit this type of development in test planning and execution, by presenting a motivational example, a formal model of test iteration and a discussion of tools to enhance the efficiency of testing when an iterative approach is used.

Key Words: iteration, spiral model, audit, evaluation, test planning

1. Introduction

Most computer software in use was not conceived in its current form. Software is developed in an evolutionary fashion via a series of modifications and releases. Boehm[4] and others have accentuated this iterative nature of the software construction process. But whether Boehm's spiral development or the more traditional waterfall development is used, the software life-cycle is always highly iterative. However, most models of software testing and verification have either neglected this iteration altogether (e.g., Morell[7]), treated it in a vague fashion (e.g., Beizer[3]) or confined consideration of it to regression testing, which evaluates if known software problems have been corrected. Virtually all models of iteration in software development have omitted discussion of its impact on test design (e.g., Luqi[6]).

This "memoryless" view of software testing ignores the fact that much testing takes place in familiar environments. A tester evaluating successive versions of a piece of software knows much about the object under evaluation. First, all versions of a piece of software share the same general

1. Professor Shimeall's participation in this research was supported by funds administered by the Naval Postgraduate School Research Council.

purpose, and much of the specific functionality geared to achieving that purpose. While there may be wide-spread modifications to the source code between releases, much of the code will be unmodified. Second, the process of version preparation will be similar from release to release, usually involving familiar operating systems and support tools, many of the same technical and managerial personnel, and established coding standards and practices. Third, the population of users and their expectations will be increasingly known to the test personnel. All of this suggests that the history of a software artifact be incorporated in the process of test design for that artifact.

The purpose of this paper is to describe a testing history and detail how it can be used to plan and execute software testing in an iterative fashion. This is done by proposing a model of the testing history of a piece of software, and illustrating the model by tracing the history of a realistic but fictitious software project. Following the example, this paper describes tools that could exploit the iterative nature of testing to make it more efficient.

2. Basic definitions

Before proceeding with the example, a few definitions of terms used in specific ways are in order. An *iteration* is one repetition of a process that is done repeatedly, most frequently the software development and testing process, in this paper. A *fault* (or, synonymously, *bug* or *problem*) is an accidental condition that causes a functional unit to fail to perform its required function. A fault may ultimately cause a *failure*, which is the inability of a system or system component to perform a required function within specified limits. *Testing* is the process of exercising or evaluating a system or system component by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results.[1]

In this paper, testing is not restricted to execution-based observation of software or system behavior, but includes *audits* and *evaluations*. The term 'audit' is used to indicate examining the program, either by execution or source code review, to test the incorporation of required functions and necessary modifications. The term 'evaluation' is used to indicate a technical review of the design or the test. Evaluations may include review of the design, test plans and test results, as appropriate.

In general, a test iteration may be modeled as a triplet, $I = (S, T(S), E(T, R, S))$, where S is a model of the subject (e.g., piece of software) being tested, T is a model of the set of test conditions (e.g., sets of input data, observation procedure or selection criteria) and E is a model of the evaluation standard (e.g., sets of expected results, observation goals, or rules to indicate acceptable implementation). A test history may be modeled as a sequence of triplets describing test iterations for a particular project,

$$H \subseteq \{ (S_i, T_j(S_i), E_k(T_j(S_i), R_{\max(i,j,k)}, S_i)) \mid 0 \le i \le n, 0 \le j \le m, 0 \le k \le l \}$$

where n is the number of revisions of the source code, m is the number of revisions of the test conditions and l is the number of revisions of the evaluation standard.

There are two ways in which the subject may be considered. A black-box test subdivides the subject solely according to its externally-visible functionality. A glass-box test subdivides the subject according to its implementation structures. The choice of glass-box vs. black-box testing affects the manner in which the test conditions are selected and the amount of detail considered in the evaluation standard.

The test conditions are often driven by specific concerns related to the test subject. The goal of the test is most often expressed via the selection of test conditions. The evaluation standard is often generated based on the test conditions, but occasionally limits on the availability of expected results will force selection of test conditions to achieve fixed results. If expected results are achieved under the test conditions, the test subject passes the test. If the expected results are not achieved under the test conditions, the test subject fails the test. If the test does not resolve the specific concerns that drove the test conditions, then the test fails. If the test resolves those concerns then the test succeeds. Subject passage or failure of a test is orthogonal to test success or failure (see Figure 1).

	Test Succeeds	Test Fails
Subject Passes	Acceptable Behavior	Vacuous Test
Subject Fails	Fault Found	Over-restrictive Test

Figure 1: Test Success/Failure vs. Subject Passage/Failure

3. MAILER: An Example Iterated Software System

3.1. Initial description

ABC Applications, a (hypothetical) company specializing in business and database applications for PCs and LANs decides to produce MAILER, a mailing label printer. It is initially intended to run as a stand-alone application, to be marketed to small- to medium-sized businesses. The program is to be able to compile a large number of addresses, along with some data elements that the user will use as a means to select or sort the addresses in the database. The software is to run in a PC environment. The MAILER Development Group (MDG) is formed to handle the analysis, design and implementation tasks. This includes unit test, product integration, configuration management and integration testing.

Functionally, the MDG organizes the software into six components:

i) The Data Definition Program, used to define the data types to be used with each customer record, data entry screens to be used to enter the data, and reports to be generated.

ii) The Database, containing the mailing list information. In order to simplify the design, ABC's existing (off-the-shelf) database package is used for this.

iii) The Data Entry Driver, the means by which mailing list information is entered into the system.

iv) The Report Generator, which displays summary information based on user queries and also generates the mailing list output.

v) The Query Tool, used to provide some basic statistical information to the user and to allow the user to select groups for mailing list generation.

vi) The Front-End Application, a menu-driven program used to access all of the other components of the software.

3.1.1. Formal Model of Test Subject

Since testing focuses on the behavior of the program along sequences of operations, testers often model a test subject (e.g., piece of software) as a graph S. Such a graph model may have nodes representing program operations and arcs indicating possible sequences of operations. For test modelling, nodes may be annotated with the pre- and post-conditions of execution of the

operations, and the arcs annotated with the boolean conditions to be satisfied before the arc may be traversed. In the case of MAILER, the initial model of the subject would be a graph with six nodes (for the components identified by the MDG), linked by arcs indicating possible flows of control between the components.

Over the course of development, the software versions released to testing may be modelled as a series of subjects, modeled by elements of the series $S_0, \ldots, S_n$, where S_0 represents the initial release and S_n represents the final retirement release. For $i > 0$, it is assumed that release i, modeled by S_i, is generated from one or more releases j, modeled by S_j, where $j < i$, but j is not necessarily $i - 1$. In other words, the sequence $S_0, \ldots, S_n$ represents a flattened hierarchy rooted at S_0.

3.1.2. Initial Test Design

As the product is designed and implemented, the Mailer Test Group (MTG) is formed. The formation of this group stretches out over the implementation of the product, and includes at least one test engineer who is brought in early in the design process to insure testability of the product and plan for later testing. As the integration progresses, and the complexity of the design product increases, the test group is increased to provide adequate expertise and to prepare for the testing effort.

This group is targeted at functional and stress test of the integrated product as it nears completion and through final product release. Secondary goals of the group include providing test expertise to the design organization, providing commentary on the design to ensure later testability, and ensuring that the overall product complies with customer needs and intended use. As part of the commentary in the area of testability, a tool is created to record operator input to the software, easing reproduction of test results. Additional design changes are made at the request of the design organization to increase the reliability of the software product.

During design and early implementation of the software, the focus of the test group is test design. The design of the test begins with an investigation of the functional design of the software and its human interface. The research into the product is then broadened to include information about intended use and also about how customers use similar or previous products in the market. The

test group also researches the development history of the product. The direction of this research is to identify areas that have had substantial problems, that are complex or risky, or that have other characteristics that would call for special attention.

Based on their research, the test group develops an initial plan for three tests of the software:

i) A functional audit, designed to exercise the software in general, and to provide a solid overview of the software operation and human interface. The purpose of this audit is not to exercise every last detail of the software, but rather to ensure that each of the components of the software is working properly on their own and in the context of the system as a whole, that the interfaces between the software and its environment are functioning properly, and that the product will meet the needs of its intended market.

ii) A stress test, designed to enter the maximum number of addresses that the database will hold and to see what happens when the user tries to enter too many addresses into the system or retrieve all addresses in a maximum database.

iii) A detail test targeted at two goals: to examine important functional areas in detail and to examine areas where problems were found during the functional audit and stress test. If this was a product with a critical application (e.g., a medical product) it would be in this area that the critical functionality would be exercised in detail.

The difference between the detail test and a regression test lies in the intent of the test. In a regression test, the goal is to determine if old bugs have been fixed. In the detail test, the goal is to locate new bugs. The evaluation does this by identifying areas with a high potential of problems and exercising them as thoroughly as possible.

Product evaluation meetings are planned, which will consist of technical reviews of the product and its progress through testing, including the potential impact of the located problems on the product. The meetings consist of appropriate designers, testers and representatives of marketing and management. The primary purpose of these meetings is to insure feedback from the testers to the designers about the product and feedback from the designers to the testers about the test. Along with informal communication between individual testers and designers, these meetings help to achieve open communication between these two groups.

Prior to release of the product to the field, it must be approved by the Product Quality Evaluation, which is a management-level meeting with representatives of the design, quality assurance, test and marketing groups. At that meeting, results of testing and the product evaluation meetings are presented to management and the release decisions are made.

The individual tests are planned in terms of goals for the test (completion criteria) and functional/ organizational areas (test areas). The individual test iterations are documented as a set of checklists identifying the tests to be covered in detail, with appropriate notes. In addition, the testers will also use the user manuals and specification/design documents as they are available. When a test confirms a specific requirement out of the software specification, the test will be annotated to indicate the requirement which has been verified.

3.1.3. Formal Models of Test Conditions and Evaluation Standard

The test conditions describe the goals, assumptions and techniques used in the testing of the test subject. One set of test conditions corresponds to one execution of the software during the test execution or one review of the software prior to, during or following execution. A test design effort is the selection of a set of test conditions, modeled as a predicate $T(S)$, and their corresponding evaluation standards, modeled as a predicate $E(T(S), R, S)$.

Each condition seeks to resolve some aspect of an issue regarding the software (i.e., each condition has a purpose). Most typically, this issue is the presence or absence of certain sorts of program faults, accessibility or inaccessibility of certain program operations, or provision of operation within stated time and space limits. More generally, the issue motivating the purpose of a test condition may involve aspects of any identifiable property of the software. The specific aspect of the issue chosen purpose of a test condition dictates execution or examination of a (possibly unary) set of paths through the test subject to determine if a set of pre- and post-conditions are satisfied. The details of execution or examination of this set of paths are obtained by application of a testing technique.

The test conditions are iteratively derived to resolve the issues that the customer (or an appropriate regulatory agency) desires to have explored. This produces a series of test condition sets, modeled by elements of the series $T_0(S), \ldots, T_m(S)$, where $T_0(S)$ models the initial, example

or customer-specified set of test conditions and $T_m(S)$ models the set of test conditions at the time of software retirement. The ith condition set, modeled by $T_i(S)$, where $i > 0$ will have one or more sets of test conditions that were used to derive the test conditions in the ith set. Note that there are n software releases and m versions of the test condition sets. In many systems n will not equal m, although it cannot generally be predicted if $n > m$ or if $n < m$. In some situations, customer oversight may choose to tightly connect the iteration of the software and the iteration of the test conditions, yielding $n = m$.

In general, the evaluation standard is a Boolean-valued function of the test conditions, results and the test subject. This Boolean function may be further subdivided into functions evaluating the internal consistency and correctness of the results, and those functions further subdivided to evaluate with respect to the design structure (i.e., are the set of results those produced by valid paths through the design structure of the software?), the designed functionality (i.e., are the set of results a valid expression of the semantics contained in the design structure?), the user (i.e., are the set of results reflective of the desired behavior for the software?), and the development history (i.e., are the set of results free of problems identified earlier in the development?).

As the test conditions and design are revised, so is the evaluation standard. This produces a series of evaluations standards modeled by elements of the series $E_0, \ldots, E_l$, where E_0 represents the expectations for the initial release under the initial test conditions and E_l represents the evaluation standard in place at software retirement. If the iteration on test conditions is not tightly connected to the iteration on software releases, it can be assumed that $l > n + m$ and otherwise $l = n$.

3.1.4. Test of Mailer

During the functional audit of the software, problems are identified in four areas. Some problems are observed in the data definition program, especially in trying to define some combinations of data types for use in later queries. There are some problems in the database interface, and a crash in the report generation when many conditions are put on the group selection.

When the problems are discussed with the designers, the designers explain that the problems with the data definition program were due to limitations on the underlying database, and show where the documentation shows how to overcome these problems.

The planned stress test is abandoned because it concentrates on the number of mailing list entries, and problems were observed in the functional audit that dealt with queries and report generation. A new stress test is produced, emphasizing the two identified problem areas by putting more conditions on the selection and by adding more report generation. This new stress test is then performed.

More problems are identified during the conduct of the stress test, and the decision is made to terminate testing on the version due to the number and severity of the already identified problems. The reason that testing is not continued to find more bugs is twofold: first, the general functional and structural areas have already been covered; second, the changes that will be required to fix the known bugs are sufficiently large that detailed testing will be invalidated.

3.2. Planning for Iteration

During the initial software test, the developers and testers learn much about the operation of the software and about means to evaluate that operation. This learning should be reflected in the future conduct of the software life-cycle for this project. In terms of test design, this means that many of the assumptions made during the initial test planning are revised as the development progresses. This revision takes the form of identifying areas of the software that require intensive testing in the next iteration, identifying redundant or unneeded tests, and varying satisfactory test cases to prevent implementors from coding to the test rather than to the required or designed behavior.

There are many possible means to identify portions of the software that warrant intensive testing in the next iteration. Three means that are often profitable are history, complexity and accessibility. Using the history of a software product to focus additional testing effort includes consideration of where source code problems have been identified in the previous iteration or iterations. If there have been recent changes to the software, the testers consider possible effects of these changes and use these effects to guide testing.

The process of testing a piece of software may add insight as to how it may be used by its intended customers. Assumptions that certain operations will be used independently may be revised and the resulting command paths verified in new test cases. As the software is moved into new markets, the users may desire additional features (or the removal of some features). As customers grow more used to the software, they may start using the software in new ways. An example of this are the 'power users' vs. the 'conventional users'. The power users often make unexpected demands on the software. As the product becomes widespread, the population of users will exercise the software much more thoroughly than the testers are able to. As such, they may identify unexpected faults, possibly involving long-duration execution of the software. An example of this is the 1991 Patriot Missile intercept failure where the control software was being operated for a longer duration than the specification anticipated.[2]

As the software is initially tested, the testers may note portions of it that are particularly complex, or that appear to have been particularly difficult to implement. These portions may be further explored in later iterations. Empirical studies have noted significant correlations between complex places in the software and the presence of faults in those places. As an example of this, the behavior of the MAILER report generation module warrants further testing.

The unit-level testing done by the development personnel often focuses on areas of the source code that closely relate to the results produced, or that are executed by a broad set of data values. This practice ignores faults that may occur in hard-to-execute portions of the software. Testers may examine the record of unit testing performed by the developers and plan their test cases to target unexamined paths.

In addition to modifying test conditions, the evaluation standard may need to be revised as the test team's initial assumptions of proper program behavior are checked and refined. Ambiguity and incompleteness in the specification and design may give the testers an incorrect or incomplete view of the proper program behavior. Testers are often forced to assume some behavior is correct (or incorrect), and to modify the assumptions after test results are reviewed with the development team and the customers. As an example, the behavior of the MAILER database module forced revision of the evaluation standard after review with the development team.

While some features of the software will need increased or maintained testing on a particular iteration, other features may not need to be tested at all. The means for identifying redundant or unneeded tests include analysis of what unchanged functionality has already been thoroughly examined, identification of tests that are subsumed by other tests, identification of functions that have been defeatured or replaced in the newest software release and identification of tests that are based on rejected assumptions of software use.

The rationale for varying satisfactory test cases is that as problems based on the initial test cases are identified to the designers and corrected, it is expected that the designers will use the test data to confirm that the problem is fixed. This use of the test data can result in situations where the code is unintentionally designed to pass the test. Furthermore, minor variations in the test case may uncover related problems that the designers did not correct or problems caused during the correction. Finally, minor variations can increase the likelihood of increased code coverage over the life of the test. To support this modification of the test cases, the test procedures and the criteria used for accepting test procedures must be designed with sufficient flexibility to allow variation in the test execution without the overhead of modifying the test documentation.

The test will be revised as the testing progresses to reflect the progress of the design, the test team's increased knowledge about the system under test, and the correction of vacuous or overly-restrictive tests. Revision of the test involves revision of both the test conditions and the evaluation standards. As the test iterations progress, some tests are refined to better insure that the software meets newly identified expectations. Some tests are added to better check problem areas, and other tests are reduced or eliminated as the areas they check become reliable and unchanging. The intent of these revisions is to provide more closely targeted test cases with greater economy, and to identify more subtle undesirable behavior than is possible on the initial iteration.

There are several factors that may cause the termination of an individual iteration. Managerially, it may be necessary to allocate a limit for resource expenditure during an iteration and to terminate an iteration when that limit is reached. Testing of a version of the software may be terminated by the presence of a new version that is desirable to replace the current version, forcing a new iteration of test planning to commence immediately. Testing of a version of the software may

reveal problems that are so serious that further testing is pointless (in the worst case, such as serious problems with data input, further testing may be impossible). The most desirable termination of an iteration, however, occurs when evaluation of the test results determines that the current version of the software is acceptable for release. The last condition under which an iteration may terminate is if the decision is made to retire the software.

3.2.1. Planning for iteration of MAILER

In planning for the next iteration of testing of MAILER, care is taken to evaluate the impact of the changes between versions. Since these areas are undergoing substantial changes, they are prime candidates for problems. In addition, it has been established during the prior testing that they already had problems, so it is unlikely that all of their problems will be fixed in a single iteration of the code.

Review of the test results from the functional audit prove useful as well. The database was exercised thoroughly during both the functional audit and the stress test, and performed well, except for documented weaknesses. No unexpected or unacceptable problems were observed, and the interface to it worked well. A substantial amount of time was spent exercising the database in both tests. For these reasons and because it will be both unchanged and outside the scope of the changed areas in the next version, the testers decide to minimize the exercise of the database in the functional audit of the next version. The fact that this is an off-the-shelf package that has already matured and been in wide use helps with this decision.

The front-end application is also considered. It is a simple menu program designed to provide the user with a 'friendly' way of accessing the different parts of the system. It has also passed the functional audit and the stress test. The decision is made to reduce testing to a simple check to make sure that it executes the appropriate programs in response to user selections. Again, the fact that it will not be changed and will be outside the scope of the changes leads to this decision.

The decision is made to emphasize the query and reporting areas, and a number of test cases are defined that will exercise both of them. In addition, a canned database is defined that will support those test cases and provide for more stress on the parts of code that are considered likely to contain faults.

3.2.2. The second iteration of MAILER

After due time, the developers indicate that the software is ready for another evaluation. The abbreviated functional audit indicates that most of the previously identified problems were fixed, but that in fixing the problems a number of new problems have been introduced in the query processing code. In the process of performing the stress test, it is found that some of the problems that were reported fixed in the report generation area are still not working properly, and some new problems are identified in those areas as well.

A detail test is done that concentrates on the query and report generation areas. In addition, the data definition area is covered in some detail, since it was within the scope of the query problem fixes. In the process of exercising the data definition area, two problems are identified in that code. Further exercise of the area is pursued as a result of the problems found there and a new problem area is identified within the data definition program. The new problem area contains a group of seven related problems, most of which are of a fairly minor nature. Two of the problems are significant enough to require correction before release to a customer.

At the PQE completing the second iteration, it is decided to fix the known problems in the query and report generation areas and the two significant problems in the data definition area. The developers report that several of the problems in the query tool will require major efforts to fix. Management decides that the problem area in the query tool is not critical to a first release and that the entire section of the query tool is disabled for the first release. The report generation area is deemed a "must fix" issue, and with the resources freed up by disabling the query tool problems, it is rapidly corrected.

3.2.3. The third iteration of MAILER

In planning for this iteration, the test designers observe that the overall level of change to the system has been significant. In addition, this version is targeted as a possible release to the field. With these factors in mind, the test designers decide to repeat the full functional audit from the first iteration, modified to look for the functional bugs found in the previous two versions and with the database area again emphasized. The reason for reemphasizing the database is the problems found in the data definition section and the possibility that the interface to the database has been affected by the changes to correct those problems.

The functional audit reveals that some of the minor problems with the data definition tool are still there. However, two of them were corrected in the process of fixing the two critical problems, and no new problems have been introduced. The database still looks good, and the interface to it has not been affected by the changes to the data definition area. Some problems still remain in the query area, but these are sufficiently minor that it is felt corrections to them can be delayed until after release.

In the stress test, it is confirmed that most of the report generation problems have been fixed, and again the problems that remain are deemed acceptable for release. No new problems are identified.

During the detail test, the problem areas are again gone over in detail. The area of the query tool that has been disabled is confirmed to be inaccessible to the user and it is confirmed that the rest of the query tool is adequate. The report generation problems are repeated and confirmed and still evaluated as acceptable. The data definition problems are confirmed fixed. In the PQE it is decided that the product is acceptable, and the product is released.

3.2.4. Initial MAILER release

The product is released and is moderately successful. However, when it is shown to several of the company's larger customers, they insist that it be interfaced to the company's existing accounting packages so that they can include data from the payroll, general ledger and inventory systems. It is decided that providing a version that interfaces to the company's existing products is a good idea, and management asks the software designers to provide a second version of the software that does this. In addition, reports of problems have come in from the field. New customers do not like some parts of the data entry driver, and a decision is made to upgrade the front-end application.

While the product was being shipped, the developers were correcting the known problems in the first release. With the added requirements, it is decided that the current crew will not be able to get the new product out in a timely fashion, so some new people are brought in to speed things up. In addition, an expert from the accounting group is added to the team to help with interfacing to the accounting and payroll packages. Also one of the original team quit and had to be replaced.

The test designers take all of this into account in planning the next iteration:

i) Instead of a single version in each new release, there will now have to be two: one for the stand-alone program and one for the accounting system. The changes to support this are substantial, and the two separate versions will have to be stress tested independently.

ii) Due to the environment on which the software is being developed, most of the code modules used in the two versions are common. Thus, most of the functional audit on one of the two versions also applies to the other version. Only the unique code and its scope of effect in the derivative version needs to be functionally audited if the auditing on the root version is adequate.

iii) The new members of the team are not as familiar with the design of the software as those who originally implemented it. They are consequently more likely to induce errors in the next iteration.

iv) The new interface to the accounting and payroll packages is a complex one. Only the designer from accounting fully understands it, and he is not familiar with the mailing

list application, thus it is very likely to have problems.

v) When the original programmer was lost, he took a certain level of knowledge about his code with him that cannot be replaced merely by reading the documentation or studying his code. This area therefore has a high potential for problems.

With these factors in mind, a new functional audit is designed that includes the new software version and emphasizes the areas with a high potential for problems, as described above. In addition, the parts of the code being worked by the new members of the team, and the part of the code that belonged to the designer who quit are to be emphasized in the third evaluation.

The stress test is overhauled and a new stress test is added for the accounting and payroll-package version. Some help in this area is found by examination of the testing on the packages themselves. A database suitable for use is acquired and the records of the testing provide information about problem areas in the packages that might affect the mailing list product.

Finally, the detail test is considered. It is planned to emphasize the areas in which there have been problems in the past and also the areas that are produced by the new designers. This is in anticipation of possible problems due to inexperience with the code.

3.2.6. The fourth iteration of MAILER

During the functional audit, the problems with the query function appear to have been resolved and the function looks like it is working well. In addition, it is confirmed that the problems with report generation have been corrected. The query facility that was disabled on the first release is retested and confirmed to be working properly.

The initial stress test of the stand-alone version of the software goes well. The stress of the accounting subsystem indicates serious problems when extreme numbers of entries are used from the various accounting master files.

The detail test of the software indicates that the new programmer was not fully aware of some of the interfaces between modules in the source code. Old problem areas are exercised and found to be in good working condition.

3.2.7. Termination of MAILER iterations

In due time, after other iterations and releases, a decision is made to abandon this software in favor of an entirely new product. A final iteration is produced and evaluated. It passes and is released to customers.

With the successful release of the final software version, the testing of this particular software item is terminated. However much of the data can be retained. This especially relates to histories of interface problems from software tools used in this development and that may be used in other development efforts. Information about user environment, requirements and developers should also be retained in case it applies to future products. In time, much of this information will be recycled and used on future product test efforts.

4. Tools to support history-based testing

The iterative nature of the life-span of a piece of software means that much of software testing takes place in an environment where much is known of the previous failures of the software. These previous failures may provide a context for improvements of the test data selection, the test execution and the evaluation of the test results. While no current software tools exploit such a context, it is quite feasible to construct such tools. This section surveys potential tools and the means used to exploit a test history to improve the efficiency and efficacy of the test effort.

One direct application of the test history would be a system to describe previously detected faults in a manner that permits them to be applied to similar programming constructs. Mutation testing[5] tools have libraries of program modifications that theoretically lead to detection of program faults. A system of notation to describe detected faults as similar program modifications would permit mutation-type testing to be applied using detected fault-related conditions, rather than hypothesized fault-related conditions. Essentially, the tester would analyze each erroneous statement to determine the specific modification that lead to the fault, and construct a statement modification pattern that could replicate such a modification elsewhere in the software. As in mutation testing, these modifications would guide selection of test data.

It may be possible to apply this sort of mutation-type testing in the absence of a test history, using pre-existing catalogues of software faults (e.g., the one in Beizer's textbook on software testing[3]). But these catalogues are often very generic descriptions, without enough detail to indicate when faults may be applicable to an application. The effects of these faults on the system state are rarely identified. Even if enough detail is present, the faults in the catalogue may not be representative of the faults in the system under test. This non-representation may lead to less complete testing than use of a test history would provide, as faults found during previous iterations are not included in the mutation generation. Use of catalogued faults may also lead to less efficient testing as inapplicable faults are considered. Thus, a test history is to be preferred over a fault catalogue as a basis for this type of testing.

One difficulty with the use of mutations produced by faults is that not all faults are isolated to a single statement. This difficulty might be overcome by constructing patterns of variable-value modifications, rather than statement modifications. To construct this sort of pattern, the tester would analyze each detected fault to determine the conditions that lead to the fault affecting the output (i.e., the sort of data lead to detection of the fault). This sort of analysis has already been performed under laboratory conditions[9], but may be too time consuming for practical use when performed manually.

Once the patterns are constructed, tools may analyze existing programs for the presence of code or data objects satisfying the patterns. Systems already exist to scan programs for complex and generic patterns (e.g., CESAR[8]), which might be adapted to this use. This type of tool would increase the effectiveness of testing by allowing the detection of groups of faults with minimal additional human effort.

Another benefit gained by exploiting an iterative test history would be the recognition and elimination of unproductive tests. Vacuous and over-restrictive tests (i.e., tests that fail to resolve the issues that lead to the design of the test) may be identified and analyzed for why they are unproductive. The results of this analysis can be used to revise or eliminate further tests to improve their efficacy. Currently this is done either on a test-by-test basis (resulting in a lot of extra work) or on a high-level modular basis (as in the preceding MAILER example).

Looking beyond modification of current testing techniques, the concept of iterative testing points to a revision in the testing process itself. Current test practices are geared to step-by-step procedures, carefully predefined, and aimed at validating predefined requirements. Testers are rarely at liberty to vary from these procedures based on factors observed during the conduct of the test. An iterative test strategy would use the predefined procedures as a starting point, but the test would be designed to allow the testers to explore fully problems noted during the test, to modify the established order of test execution based on changed perceptions of the requirements, or to modify test conditions based on increased understanding of the usage environment. The goal would be to focus the test less on accomplishing a predefined procedure and more on resolving a set of issues regarding the software and its behavior.

Since, under the iterative view of testing, the test process may be applied more than once to a given version of the software, it may be wise to reconsider the canonical view of testing only when implementation is complete. A test effort in parallel with implementation, evaluating isolated subsystems prior to complete software integration, may accelerate the development process. Testers may note systematic errors in the design and implementation of the software, identified early enough so that management may revise the design and implementation practices for the remaining subsystem to preclude these errors. Parallel efforts may also allow more rapid test reporting than current methods, so that the needs of the implementation organization and the customer organization are fully met.

5. Summary and Conclusions

Software is rarely developed to its released form in one sequence from requirements through implementation. Rather, the initial release of a piece of software is developed through iteration among the tasks of analysis, design, implementation and verification. The history of a piece of software after its initial release is a series of further iterations through these tasks. The iterative nature of software is widely recognized among software implementors (in particular, in the rapid prototyping community), but has little impact in the current planning and conduct of software tests.

This paper has proposed a structure for representing the history of a piece of software. The history is modelled as a series of triples, the elements of which represent a version of the software, the test conditions to be applied to that version, and the standard by which the test results are to be evaluated. At each point in the life-span of a software project, one or more of these elements are being revised, as modifications are made to the software, the usage conditions are better captured in the test conditions, and the desires of the users are more completely expressed in the evaluation standard.

The paper then justified this model by tracing a realistic sample software project, noting how the version, test conditions and evaluation standards are modified during its life-span. Much of the information needed to effectively test that project was not available until after the start of testing, forcing frequent, and occasionally large, modifications of the initial test plans. Events and observations from an iteration of the implementation/testing cycle were used to plan and implement succeeding software versions and tests, at times forcing immediate revision of the software, its test conditions and its evaluation standard. The history of projects such as the example suggest that testing needs to be managed and conducted as a flexible activity, able to respond on-the-fly when events and observations warrant.

The final contribution of this paper was a discussion of how the history might be used to make testing more efficient and effective across the life-span of a software project. Techniques were described that use the test history as a source of information to revise both the software and the testing that verifies that software. The principal conclusion to be drawn from these techniques is that testing cannot be conducted most efficiently when the test procedures are viewed as predefined and immutable. Rather, if the goal is to produce the most reliable software for the least cost, testers need to be free to exploit information gathered during a test in the conduct of that test and in the planning and conduct of future tests.

6. Acknowledgments

The authors wish to thank Richard Hamming and Richard Shimeall for their many helpful comments.

References

[1] *Glossary of Software Engineering Terminology*, ANSI/IEEE Std. 729-1983, IEEE Computer Society, 1983.

[2] "Patriot Missile Defense Software Problem Led to System Failure at Dhahran, Saudi Arabia", Technical Report GAO/IMTEC-92-26, General Accounting Office, Washington DC, 1992.

[3] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1983.

[4] Boehm, B. H., "A Spiral Model of Software Development and Enhancement", IEEE Computer, May 1988, pp. 61-72.

[5] DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer*, April 1978, pp. 34-41.

[6] Luqi, "A Graph Model of Software Evolution", *IEEE Transactions on Software Engineering*, Vol. 16, no. 8, 1990, pp. 917-927.

[7] Morell, L.J., "A Theory of Fault-Based Testing", *IEEE Transactions on Software Engineering*, Vol. 16, no. 8, 1990, pp. 844-857.

[8] Olender, K. and Osterweil, L., "Cesar: A Static Sequencing Constraint Analyzer", *Third Symposium on Software Testing Analysis, and Verification*, December 1989, pp. 66-74.

[9] Shimeall, T. J., Bolchoz, J. M., and Griffin, R., "Analytical Derivation of Software Failure Regions", Technical Report NPSCS-91-001, Computer Science Dept., Naval Postgraduate School, 1991.

Distribution List

Defense Technical Information Center Cameron Station, Alexandria, VA 22314	2 copies
Library, Code 0142 Naval Postgraduate School, Monterey, CA 93943	2 copies
Center for Naval Analyses 4401 Ford Avenue Alexandria, VA 22302-0268	1 copy
Research Office Code 08 Naval Postgraduate School, Monterey, CA 93943	1 copy
Dr. Timothy Shimeall Code CS/Sm, Dept. of Computer Science Naval Postgraduate School Monterey, California 93943-5100	15 copies
Stephen Shimeall Boeing Defense & Space Group Aerospace & Electronics Division P.O. Box 3999 M/S 1P-13 Seattle, Washington 98124-2499	5 copies